Lab 6 Cache Lab Report

Name: Amanuel Assefa 20170843
Eyuel Elala 20170862

Introduction

In this assignment we added a cache on top of the pipelined microprocessor which we designed in the previous lab. There some disadvantages of pipelined implementation without cache on the performance as memory access is expensive in real systems. So in order to improve the performance we have used some cache to increase the performance.

Design

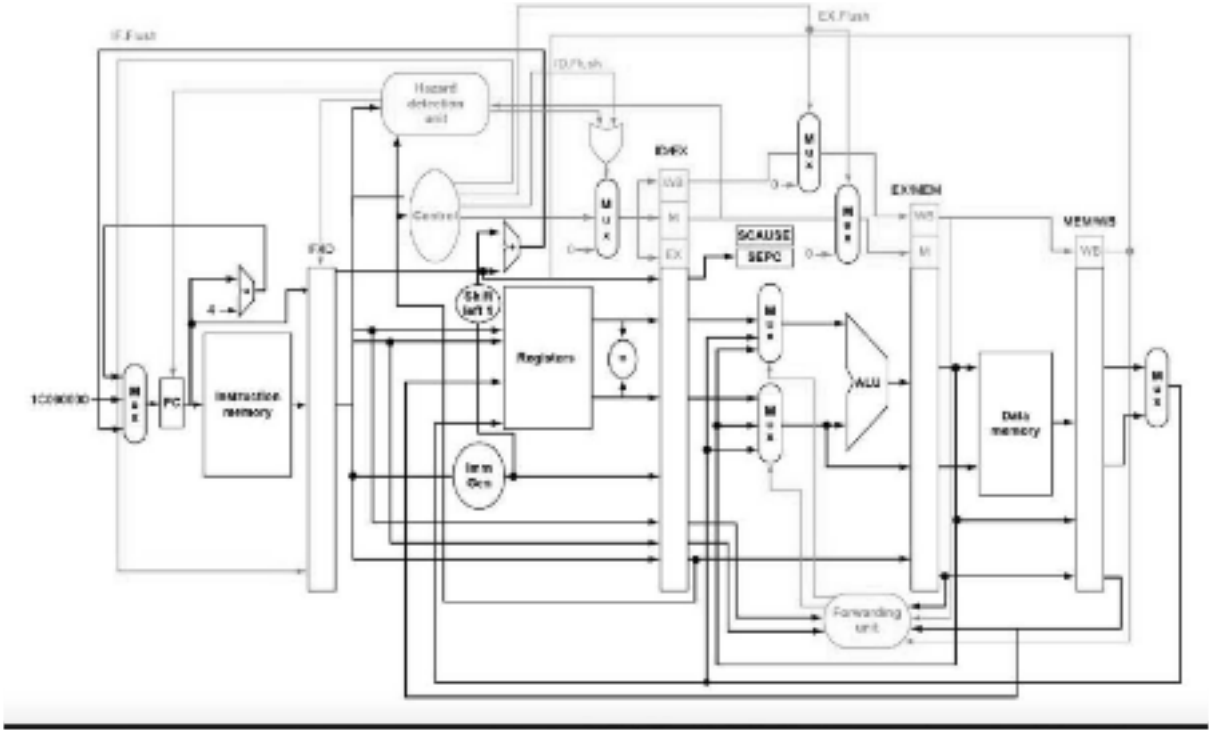

Fig1. Next to the data memory there is a small cache where load and store instructions access cache first

Fetching stage

After setting the program counter , we fetch the instruction from the memory model and send it to RISC_TOP , and at the end of the clock cycle we store the control signals and all necessary information in the IF/ID pipeline register and PC+4 is fetched, because we implemented always not taken.

Decoding stage

This is the stage of identifying the instruction. The instruction is partitioned and appropriate information is extracted and all necessary control signals and information are passed to the next pipeline register.

Executing stage

Depending on the instruction, using the appropriate variable extracted from the instruction this stage make use of the ALU to get the result or output of the operation and passed to the next pipeline register.

Memory Stage **/*****Important***/**

In this stage compared to the previous lab, memory accessing instructions first access cache, then depending on it being a hit or a miss the following happens. If it is a cache hit the latency is only one cycle, so if the instruction is a store we use the "write through" write hit method, we synchronously write the new data to both the cache and memory. If it is a load and a hit, we just load the value from the cache , without accessing memory, we make the MemRead control signal equal to 0 to achieve this. If it is a cache miss, we implement a blocking cache, where the pipeline is stalled and all pipeline registers from the IF/ID to EX/MEM don't move. The pipeline resumes as normal after 6 cycles have elapsed.We do nothing in the first cycle, but starting from the second to the fifth stage we bring the 4 words from memory and store it in the appropriate cache line and appropriate block that is known from the 14 bit memory address, so we implemented the "write allocate" method. The 14 bit memory address includes the 2 bit byte offset, 2 bit block offset, 3 bit index(because there are 8 cache lines),and 7 bit tag. And the cache is a 136 bit by 8 memory structure, where 128 bits are for storing the 4 words , the 1 bit for the valid bit and 7 bit for tag.

And in the 6th cycle depending on the instruction being a store or a load, if it is a store we write the new value to both the cache and memory at the negative edge synchronously. If it is a load we don't access memory in the 6th cycle, but take the value from the cache.

Write back Stage

Data is written back to the register, based on the instruction given we determine the data .

Implementation

In the implementation part we took most part of the code form our single cycle . We used our previous ALU Module with a little modification. We implemented the ALU like this , we supply the opcode, funct3 and funct7 that we got from the instruction and according to those values we do the appropriate computation and return ALU result and bcond . Bcond is the branch condition that is on if it is a branch instruction . The control signal module is supplied with the instruction and makes the appropriate control signals for the Datapath. Even for the control signal we modified a bit from the previous part .After designing our pipelined microarchitecture, we started the implementation by writing module for handling stalling and forwarding.

stallAndForward: This module has input of three consecutive instructions and an output of stall flag and a control signal for choosing the inputs of the ALU

After designing this part, the started writing the RISCV_top file. We were writing for each stage one by one. Control signals are generated in the first stage and they are latched on the pipelined registers, so as we move to the next stage we forward them and discard some of the control signals that aren't important for the latter stages. New control signals might also be generated depending on the situation. In order to make our implementation easier

we forwarded the instruction so that we can get useful information from it by decoding. So every executed instruction will get its information from the corresponding pipeline register. Sometimes instruction in decoding stage can look at the pipeline register in execution stage to check dependency and take the appropriate action.
One major problem while implementing the pipelined CPU is that when an instruction is branch instruction we can't know next value of the program counter until the execution stage. However stalling reduces performance so we used **always not taken branch prediction for the previous lab**. Then if the result we get after the execution stage and the predicted one aren't the same we flush them and fetch the correct one. This is better than stalling for every branch instruction.
To implement the cache we used a 2 dimensional array , and we defined registers so that we know the number of hits and misses, and also other registers for unique purposes. We used the instruction register of the EX/MEM pipeline register to trigger the checking of the cache if the instruction is a load or store. And we use if else statements to determine cache hit or miss.

Evaluation

Compared to the previous pipeline lab, the number of cycles is going to be higher, because of the cache misses. We are going to evaluate the 3 files next

Inst:

In the inst file, cache was not accessed as there were no memory accessing instructions, so the number of hits and misses are zero and zero respectively, and the cycle is exactly the same as the previous lab, which is 24 cycles

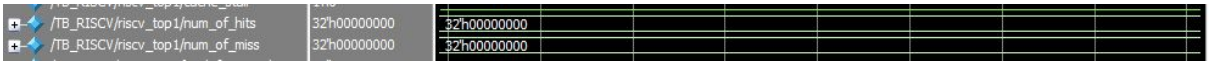

Fig 2. The number of hits and misses in inst file

Fig 3. The cycle information for inst.v

Forloop:

In the forloop file,cache was accessed. The number of hits and misses are presented below. The number of hits were 1f(in hex), 31 in decimal. The number of misses were 3. So the cache hit ratio = 31/34 = 91.18%. Since there were 3 misses, and the cache miss costs 6 cycle, each miss costs 5 more cycles compared to the previous lab, because in the previous lab mem stage latency was 1 cycle, now cache miss latency is 6 cycle, so the difference is 5 cycle. So we expect 3 *5 = 15 more cycles. In our previous lab the number of cycles was 102 cycles, now it is 117 cycles. According to the previous calculation our result is correct. And we manually checked the assembly to figure out how many misses it would have and it has 3 misses with manual inspection. So our result is correct(our guess :)).

Fig 4. The number of hits and misses

```
VSIM 6> run -all
# Finish:        117 cycle
# Success.
# ** Note: $finish    : C:/Users/amans/Downloads/TB_RISCV_forloop.v(166)
#    Time: 1285 ns  Iteration: 1  Instance: /TB_RISCV
```

Fig 5. The cycle information for forloop.v

Sort:

In the Sort file,cache was accessed. The number of hits and misses are presented below. The number of hits were 1352(in hex), 4946 in decimal. The number of misses were 2ff(in hex), 767 in decimal. So the cache hit ratio = 4946/ 5713= 86.57%. Since there were 767 misses, and the cache miss costs 6 cycle, each miss costs 5 more cycles compared to the previous lab, because in the previous lab mem stage latency was 1 cycle, now cache miss latency is 6 cycle, so the difference is 5 cycle. So we expect 767 *5 = 3835 more cycles. In our previous lab the number of cycles was 14729 cycles, now it should be 14729 + 3835 = 18564 according to our previous calculation. According to the previous calculation our result is correct. But unlike forloop, it is very hard to make a manual inspection, so we just hope our implementation is correct.

Fig 5. The number of hits and misses

```
VSIM 9> run -all
# Finish:      18564 cycle
# Success.
# ** Note: $finish    : C:/Users/amans/Downloads/TB_RISCV_sort.v(192)
#    Time: 185755 ns  Iteration: 1  Instance: /TB_RISCV
```

Figure 6 TB_RISCV_sort testbench results

***** We implemented the variables used for the cache in line 30 - 51. And we implemented the cache using always blocks from line 419 -596.****

Discussion

In this assignment we didn't have any problems, because the homework is much easier than the previous lab.

Conclusion

Generally, this assignment helps us understand the use of cache and help us have more experience in Verilog.